Tcl Quoting

Tcl quoting is elegant, minimal, and consistent. However, it takes a while to learn to see the multiple layers of interpretation that can be involved with features like evaluation, list interpretation, or other domain-specific languages understood by routines like expr, regexp, or string match. Someone new to Tcl almost always resorts initially to unnecessarily-complicated quoting schemes. The most common cause of Tcl quoting confusion is assumptions that are carried over from other languages. Tcl is not sh and it is not C. It is also distinctly different from Lisp, Perl, or Python or nearly any other language. Its design is a novel synthesis of features and ideas from sh, C, Awk, and Fortran, among others. To appreciate its syntax, be prepared for something fundamentally different, minimal, and pure.

See Also

double substitution
How Tcl command evaluation affects routines that do their own interpretation.

Tcl syntax

everything is a string
The universal data interface provided by Tcl.

Tcl Minimal Escaping Style
Advocates using braces and quotes only when necessary.

Tcl_Merge

eval
Discusses proper quoting of scripts, particularly when they are programmatically composed.

Frequently-Made Mistakes

list
The key to programatically substituting a properly-quoted word into a script.

Is white space significant in Tcl?
Covers much of the material pertinent to Tcl quoting.

Move any widget
Contains an over-quoted script and a fixed version.

string map
The go-to routine for help generating properly-quoted Tcl code

Toplevel Watermark
Contains an over-quoted script and a fixed version.

Quoting Hell
You know you're a sinner when...

Quoting Heaven!
Can't have one without the other.

Quoting and function arguments
A pre-8.6 explanation of a few gotchas.

README.programmer , by Brent Welch, 1993
When and how to use list to avoid quoting hell. Predates {*}.

Description

In a Tcl script most characters represent their literal value. There is a small set of characters, ", $, [, {, \, and white space, that have functional meaning, but that can be quoted so that they lose that meaning and represent themselves instead. Whereas in other languages quoting may specify that a value is a string, in Tcl each value already is a string, so quoting is just another mechanism to choose which characters are interpreted, and which characters represent themselves.

What Not to Do

To stay out of trouble, follow this advice:

. Passing multiple arguments slows things down considerably because a new expression is effectively being generated and parsed each time instead a single expression being byte-compiled the first time. Secondly it might result in unintended double substitution.

Don't attempt to use braces to turn a value into a list or a script. It doesn't work that way. For example, enclosing a value in braces doesn't turn it into a list:

```
set value "what does { do?"
set script "set var {$value}"

#Here comes an error:
eval $script
```

The value of $script is clearly incomplete:

```
set var {what does { do?}
```

To do it correctly use list instead of enclosing a value in braces:

```
set value "what does { do? "
set script "set var [list $value]"
eval $script
```

or even better:

```
set value "what does { do? "
set script [list set var $value]
eval $script
```

Basic Quoting Techniques

For simple values no quoting is needed:

```
puts Hello

set greeting Hello
puts $greeting
```

Substitutions can be performed on any word, including the name of the command:

```
set cmd puts
set greeting Hello
$cmd $greeting
```

A substitution is logically part of a bare string:

```
set prefix antidis
set root establish
set suffix arian
set entry $prefix[set root]ment${suffix}ism
```

No quoting is needed in the example above because the value contains no literal whitespace outside of the script substitution, and $ is intended as a variable substitution.

Perhaps, this example needs to be expanded. The *$prefix[set root]ment${suffix}ism* argument of *set* command is taken by Tcl as a whole, because it doesn't contain any spaces. The only apparent space is a part of *[set root]* command substitution and as such it isn't a real space that divides arguments of *set* command. So, firstly Tcl sees all of *$prefix[set root]ment${suffix}ism* as one argument, then Tcl performs all substitutions of it, then Tcl passes the resulting string to *set* command. We can freely use multiple words in variables' values of this example:

```
set prefix {me antidis.}
set root   { my establish}
set suffix {. my arian}
set entry $prefix[set root]ment${suffix}ism
puts $entry
# a bit longer example presenting the same one argument for Tcl:
set entry "$prefix[set root]ment${suffix}ism"
puts $entry
```

Another consideration is that, though all is done in one pass without recursions, the substitutions are performed from left to right. So, if a "left" command modifies a "right" variable's value, the result will include this new value:

```
set prefix {me antidis.}
set root   { my establish}
set suffix {. my arian}
set entry $prefix[set root { new Establish}]ment${suffix}ism:${root}ment!
```

variable substitution and script substitution can be used in a bare string:

```
set digits 245
puts 01[string replace $digits 1 1 34]67
```

output:

```
01234567
```

No quoting is required here because all characters in the script have their normal syntactic meaning. [and] indicate a script substitution, and the whitespace in that script does not affect the outer script.

When a word includes literal whitespace, use {}

```
set greeting {Good morning}
puts $greeting
```

Variable substitution and script substitution are not performed within {}, so when those features are needed, " can be used instead of {}:

```
set name Bob
proc daystage {return Morning}
set greeting "Good [daystage], $name."
puts $greeting
```

output:

```
Good Morning, Bob.
```

Composing a Script for Evaluation

First, it's often preferable to compose a single command needing no substitutions:

```
proc reputation_cb {msg msg2} {
    puts $msg
    puts $msg2
    set time [clock seconds]
    puts [list {The time is now} [clock format $time]]
    set ::done 1
}

proc reputation {} {
    set msg {an idle and false imposition}
    set msg2 {got without merit}
    eval [list reputation_cb $msg $msg2]
}
reputation
```

This is likely to be more performant, as procedures are byte-compiled.

The apply variant of this example is:

```
proc reputation {} {
    set msg {an idle and false imposition}
    set msg2 {got without merit}
    set script {
        puts $msg
        puts $msg2
        set time [clock seconds]
        puts [list {The time is now} [clock format $time]]
    }
    eval [list apply [list {msg msg2} $script [namespace current]] $msg $msg2]
}
reputation
```

To substitute a complete word into a script, use list to turn the word into a list containing only the word. This works because syntactically a command containing no substitutions is a list, so each word in the command has the same syntax as an item in a list. If a value that is not a list is substituted into a script but not quoted, the resulting script might be syntactically incorrect:

```
set msg {hello world}
set script "set msg2 $msg"
# Here comes an error:
eval $script
```

Here the value of $script is

```
set msg2 hello world
```

which results in the error message:

```
wrong # args: should be "set varName ?newValue?"
```

list correctly quotes the substituted value:

```
set msg {hello world}
set script "set msg2 [list $msg]"
eval $script
```

In this case the value of $script is what was intended:

```
set msg2 {hello world}
```

A small but important variation that is more common is to make the entire command into a list:

```
set script [list set msg2 $msg]
```

This is preferred because even though every value is a string, Tcl is careful not to generate an actual string representation for the value until something actually requires it. In this case eval can use the internal list representation of the script directly, which has two advantages: First, a string representation of the script is not generated, which saves time and memory. Second, eval doesn't have to spend time parsing the string representation of the script. Instead, it uses the internal list representation as the parsed representation of the script, which is a big win in terms of efficiency. This has a rather far-reaching implication: Even though Tcl is known as a string-based language, it is really a list-processing language, and the low-friction way to program in Tcl is to embrace a list-oriented programming style.

string map can also be used to properly escape a word substituted into a script:

```
set msg {hello world}
set script [string map [list @msg@ [list $msg]] {
    set msg2 @msg@
}]
eval $script
```

To avoid ambiguity it's a good idea to give the placeholders a distinctly visible initial and final character. This is particularly true in large scripts, where the author might overlook the fact that a placeholder without clear delimiters occurs as a substring of another value. @ is common choice for the delimiter.

In a small script list is more convenient, but in a larger script where the additional escaping of other characters makes the script more difficult to read string map is cleaner.

ycl string template provides a more concise way to use string map.

Composing a Script for Later Evaluation

When composing a script for later evaluation, consider that the current evaluation environment might not be the environment the script eventually is evaluated in. Two examples:

```
set msg goodbye
namespace eval ns1 {
    set msg hello
    proc doitlater {} {
        set msg hello
        after 1000 {
            puts $msg
            set ::done 1
        }
    }
    doitlater
    vwait ::done
}
```

Here, the scheduled script returns 'hello'. Reason is not that "set msg" is defined in "proc doitlater". This is defined locally in the proc and the code executed by 'after' is executed in the global namespace. But, there is the "set msg hello" inside "namespace eval ns1". Msg is not defined as a variable of that namespace. That means msg is the variable defined in the calling context, which is the global one.

```
set msg goodbye
namespace eval ns1 {
    proc doitlater {} {
        variable msg hello
        after 1000 {
            puts $msg
            set ::done 1
        }
    }
    doitlater
    vwait ::done
}
```

Now, we get 'goodbye'. There exists two variables msg, one in the global context and one in the namespace ns1. The script provided to 'after' is executed in the global namespace. That means the global msg is used which is set to 'goodbye'.

Instead, either use 'apply' to capture the local value of $msg, use namespace code to capture the current namespace or use the fully-qualified variable name:

```
set msg goodbye
namespace eval ns1 {
    proc doitlater {} {
        variable msg hello
        after 1000 [list apply {msg {
            puts $msg
        }} $msg]
        after 2000 [namespace code {
            puts $msg
        }]
        after 3000 {
            puts $::ns1::msg
            set ::done 1
        }
    }
    doitlater
    vwait ::done
}
```

"namespace eval ::ns1" is used to assure we know the fully-qualified name of the namespace and with this the fully-qualified name of the variable. If this is not wanted you have to stay with 'apply' or 'namespace code'.

Alternatively, apply could also be used to capture the current namespace:

```
set msg goodbye
namespace eval ns1 {
    variable msg hello
    proc doitlater {} {
        set msg hello
        after 1000 [list apply [list {} {
            variable msg
            puts $msg
            set ::done 1
        } [namespace current]]]
    }
    doitlater
    vwait ::done
}
```

Passing $args to Another Command

In the following script, $args are any additional options to button, e.g., -fg blue

```
#! /bin/env tclsh
package require Tk
proc mybutton {parent name label args} {
    if {$parent eq {.}} {
        set myname $parent$name
    } else {
        set myname $parent.$name
    }
    button $myname -text $label -command [list puts stdout $label] {*}$args
    pack append $parent $myname {left fill}
}

mybutton . hello whadda -bg aquamarine
```

{*}$args is the right way to do it, but prior to Tcl 8.5, it was necessary to use eval:

```
eval {button $myname -text $label -command [list puts stdout $label]} $args
```

Alternatively, explicitly combine the lists first rather than relying on eval to concatenate its arguments:

```
set cmd [concat {button $myname -text $label -command [list puts stdout $label]}
$args]
eval $cmd
```

Quoting at the C level

When composing an individual command to be evaluated from inside C code, use Tcl_Merge, which composes a list from *argv*. The result can then be passed to Tcl_Eval. Tcl_VarEval does not take pains to compose a list from its arguments, but simply concatenates them together. If its arguments happen to not be lists, the results could be unexpected. APN This advice is really applicable only when you have the command in

the form of *argv* to begin with. Normally, at the C level arguments already exist as an array or list of Tcl_Obj structures and you should use Tcl_EvalObj* variants to evaluate commands, not Tcl_Eval or Tcl_VarEval.

Misc

AMG: Tcl by Ian Lance Taylor, 2011-03-31, discusses Tcl and alleges that its EIAS philosophy is its downfall. He argues that EIAS requires very precise quoting in order to get anything nontrivial to work right: *"Even then I recently wrote some Tcl code with seven consecutive backslashes, admittedly in a complex use case. That's too much for easy reasoning, and in practice requires trial and error to get right."* Sounds like a case of Quoting Hell, alright. I wish I had the chance to see the code in question and suggest an alternative, since in my experience there's always been a safe, clean way to avoid Quoting Hell.

AMG: Here's pooryorick's response to the article linked above:

This post mis-characterizes most of the aspects of Tcl that it attempts to describe. In particular, the comment about seven consecutive backslashes is a strong hint that Ian didn’t grasp the elegance of Tcl quoting. This happens when people come to Tcl steeped in other language traditions, and attempt to apply those traditions to to Tcl, which is a different sort of creature. With all due respect to Ian, each of the criticisms in this article indicate that he just didn’t stick with Tcl long enough, or perhaps look into it deeply enough to resolve his misunderstandings of the language. More info at [L1].

Updated 2021-09-24 10:52:07